第2章

C関数をハードウェア化して高速化

Nios II C-to-Hardware アクセラレーション・コンパイラを試す

浅井 剛

ここでは，米国Altera社の「Nios II C-to-Hardware アクセラレーション・コンパイラ」を取り上げる．これは，ANSI Cソフトウェア・コードから，ハード・ワイヤード論理のアクセラレータ回路を生成する合成ツールである．C言語のソフトウェア・コードのうち，ユーザが指定したファンクション（サブルーチン）をRTL（register transfer level）の回路（VHDLまたはVerilog HDL）に変換する．本稿では，ツールの基本機能とアクセラレーション効果を具体的なコードを使って解説する．評価版ツールはAltera社のWebサイト（http://www.altera.co.jp/）からダウンロードできる．（編集部）

2006年4月，Altera社は，「Nios II C-to-Hardware アクセラレーション・コンパイラ（以降，C2Hコンパイラと呼ぶ）」を発表しました．

C2Hコンパイラは，Altera社のソフト・マクロのCPU「Nios II」を適用した設計において，ソフトウェア処理性能のボトルネックを関数単位でハードウェア化することにより，高速化を図るツールです．ハードウェア化したモジュールの回路への組み込みやその制御ソフトウェア，ハードウェア化する前の関数と同等に見せるためのラッパ（wrapper）まで自動的に生成してくれるので，ユーザはハードウェア化を強く意識することなく，性能向上を図ることができます．

今回，C2Hコンパイラの評価を行う機会がありましたので，使用感についてレポートします．

1. C2Hコンパイラの概要

Altera社によれば，C2Hコンパイラの主要機能は「C言語で書かれたNios II用ソフトウェアで，高速化を妨げる関数（サブルーチン）をFPGAによるハードウェア・アクセラレータに変換」するツールとなります．

● 引き数と戻り値の受け渡しのしくみ

ソフトウェアの関数呼び出しでは，関数を呼び出す上位ソフトウェアと関数との間で，必要に応じて引き数と戻り値の受け渡しがあります．関数をハードウェア化する場合には，引き数と戻り値のやりとりを行うなんらかのしくみが必要になります．C2Hコンパイラは，これを二つの方法で実現しています．

（1）レジスタ経由（スレーブ・インターフェース）

CPUの周辺モジュールは，CPUから制御を受けて動作します．このとき，レジスタ（コマンドやモード，パラメータなど）を使って制御するのが一般的です．生成されたアクセラレータ・モジュールのレジスタにより引き数と戻り値の受け渡しを実現するのがスレーブ・インターフェースです．

複数の引き数を持ち，戻り値がある関数を考えます．Cコードは，**図1（a）**のようになります．ここでは，関数内にポインタ・アクセスは含んでいないものとします．また引き数と戻り値を，すべて関数を呼んでいる上位ソフトウェアのローカル変数としています．実際にはさらに上位階

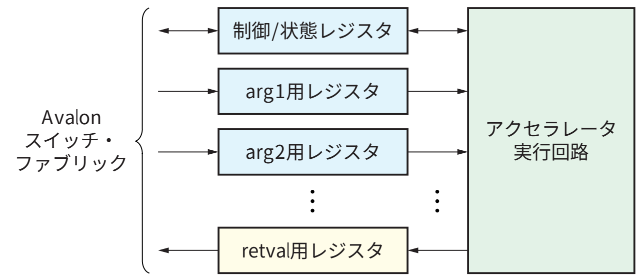
KeyWord
Nios II，C言語，C-to-Hardwareアクセラレーション・コンパイラ，関数，ハードウェア・アクセラレータ，演算子，ステート・マシン，Quartus II，EDS，Flash Programmer

```
{
 short arg1, arg2, ･･･; /* func1へ渡す引き数のローカル変数 */
 short  retval; /* func1の戻り値を格納するローカル変数 */

 arg1 = ***;     /* 引き数値の設定 */
.
.
 retval = func1(arg1, arg2, ･･･);

後処理
}
```

（a） C記述例



（b） ブロック図

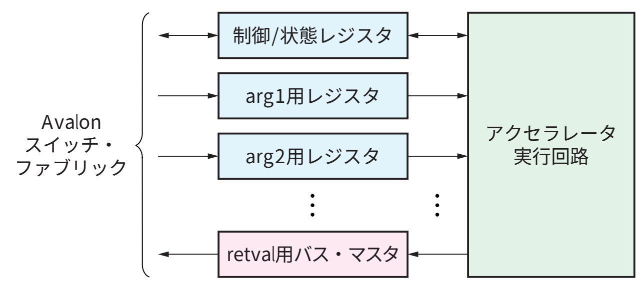
図1　スレーブ・インターフェース
複数の引き数を持ち，戻り値がある関数はレジスタ経由でアクセスする．

```
{
 short arg1, arg2, ･･･; /* func1へ渡す引き数のローカル変数 */
 short  retval; /* func1の戻り値を格納するローカル変数 */

 arg1 = ***;     /* 引き数値の設定 */
.
.
 func1(arg1, arg2, ･･･, &retval);

後処理
}
```

（a） C記述例



（b） ブロック図

図2　マスタ・インターフェース（引き数にポインタがある場合）
関数からの戻り値を引き数として指定する場合は，処理結果を指定されたアドレスへ書き込むためのバス・マスタを生成する．

層からの引き数やグローバル変数が含まれることもあります．呼ばれる関数にとっては，それらがどのような属性なのかは意識せず，すべてCPUからアクセス可能なスレーブ・レジスタとして変換されます．

C2Hコンパイラは，**図1(b)**のように，引き数と戻り値を受け渡すレジスタ群とアクセラレータ回路の起動や終了判定をするための制御/状態レジスタを生成します．これらのレジスタを介してCPUとアクセラレーション・モジュールのインターフェースをとります．

(2) ポインタ・アクセス（マスタ・インターフェース）

ポインタによるアクセスは，さらに二つのタイプに分けられます．

● 関数からの戻り値を引き数として指定する場合

関数からの戻り値を得るために，関数の引き数リストにポインタで指定することがあります．例えば，配列を戻り値とする場合に使われる方法です．戻り値の格納先がポインタ（アドレス）になります．Cコードは，**図2(a)**のようになります．

C2Hコンパイラは，**図2(b)**のように，処理結果を指定されたアドレスへ書き込むためのバス・マスタを生成します．

● 関数中にポインタ・アクセスがある場合

関数の戻り値以外に，関数中にポインタ変数に対する読み書きが含まれる場合があります．Cコードの例を**図3(a)**に示します．wptrとrptrは関数の外部で定義されているものとします．

C2Hコンパイラは，**図3(b)**のように，それぞれのポインタに対応するバス・マスタを生成します．

● 演算回路のアクセラレーション

Cソース・コードの演算子は，等価なハードウェアに変換します．Altera社によれば，演算子とは，

　*，　+，　%，　>>

を示しているそうです．

```
y = x << 3;
```

のように固定ビット数のシフトや，

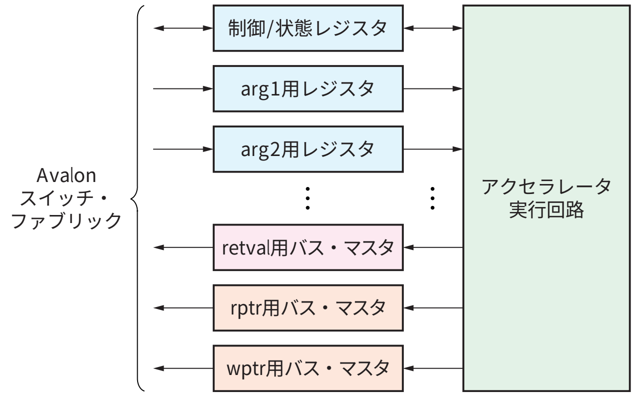
```
x &= 0xf
```

のように定数を用いた演算は，配線だけで実現され，論理リソースは消費されません．

```
func1(arg1, arg2, …, retval)
short arg1, arg2;
short *retval;
{
  short i;

  for(i=; i<10; i++)
   *wptr++ = *rptr++;
}
```

(a) C記述例



(b) ブロック図

図3　マスタ・インターフェース(関数中にポインタがある場合)
ポインタ変数ごとにバス・マスタを生成する．

● 制御回路のアクセラレーション

for文やwhile文のような繰り返し処理や，if文やcase文のような分岐処理のように，データの流れを制御する記述は，各処理に応じたステート・マシンやマルチプレクサなどの制御回路へ変換します．

図4は，C2HコンパイラによってC関数がハードウェア化されるようすを示したものです．HDLによる論理設計の経験をお持ちの方は，C言語のソース・リストを仕様書としてHDL記述していると考えれば，C2Hコンパイラがイメージできるでしょう．

2. C2Hコンパイラの準備

C2Hコンパイラを使用するためには，以下のツールが必要です．今回使用した各ツールは，執筆時点における最新バージョンです．すべてAltera社のWebサイト(http://www.altera.co.jp/support/software/download/sof-download_center.html)からダウンロードできます．

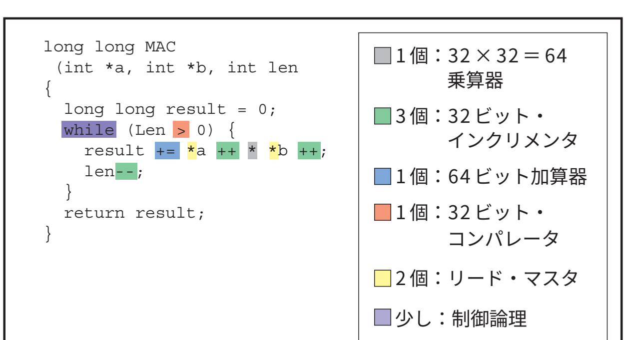


図4　C2HコンパイラによるC関数のハードウェア化
演算子は演算回路に，制御文は制御論理(ステート・マシンなど)にそのまま置き換わる．

- Quartus II Web Edition Software Version 6.0 Service Pack 1(262Mバイト)
- Nios II Embedded Design Suite, Evaluation Edition (469Mバイト)
- Nios II Embedded Design Suite Version 6.0 Service Pack 1(58.9Mバイト)

C2Hコンパイラは，Nios II Embedded Design Suite (EDS)に含まれます．EDSはQuartus IIのアドオンであり，Service Packはパッチなので，上から順にインストールする必要があります．

すでにNios IIユーザであれば，新たにツールをインストールする必要はありません．ただし，使用にあたってはライセンスの入手が必要です．評価ライセンスは，本稿執筆時点(2006年9月)ではAltera社の代理店から入手できます．いずれはほかの評価版ツールと同じようにAltera社のWebサイトから申請できるようになると思います．

● Nios II Embedded Design Suiteのインストール

Altera社のFPGAを使用していても，Nios IIをまだ使ったことのない方は，ツールのインストールを行う必要があります．ここでは，Quartus IIはすでにインストール済みとして，EDSのセットアップ手順を説明します(**図5**)．

(1) ダウンロード・ファイルの実行

Altera社のWebサイトからダウンロードしたEDSのセットアップ・ファイル(niosii_60_eval_pc.exe)を起動します．Welcome画面が表示されたら[Next>]ボタンを押します．

(2) ライセンス同意

License Agreementが表示されるので，内容を確認して，[Yes]ボタンを押します．

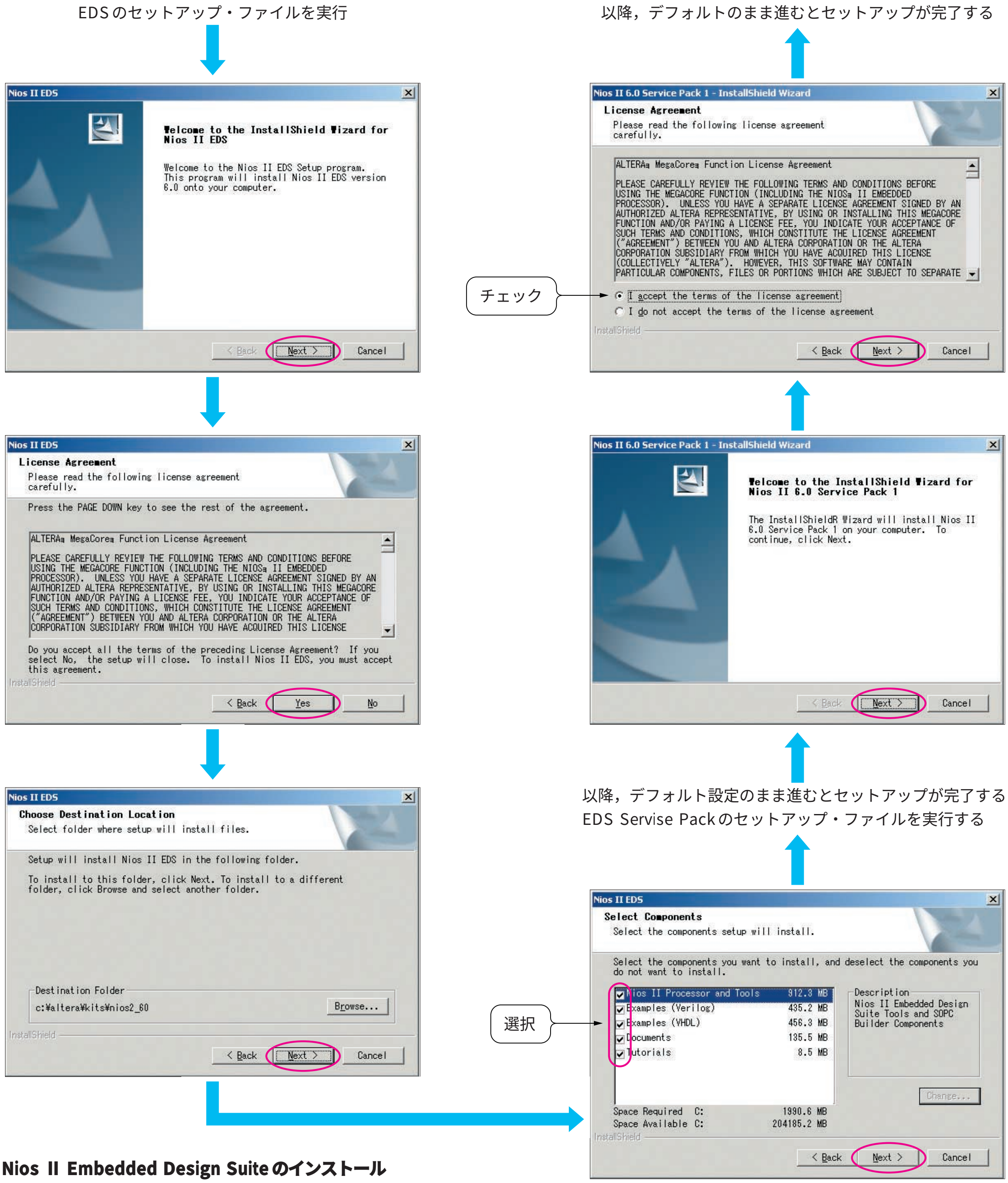


図5　Nios II Embedded Design Suiteのインストール

(3) インストール・フォルダの設定

Choose Destination Location画面では，インストール先のフォルダを指定します．ここではデフォルトのまま［Next>］ボタンを押します．

(4) インストールするコンポーネントの選択

Select Component画面では，インストールするコンポーネントを選択します．デフォルトではすべて選択されているので［Next>］ボタンで進んでかまいません．不要なコンポーネントがあればチェックを外します．HDLのExampleは，VHDLかVerilog HDLの一方だけでよい方もいると思います．

(5) Noteダイアログ

Cyclone II（EP2C35）に関する注意書きが表示されます．気にせず［はい（Y）］ボタンを押します．

(6) スタート・メニューの設定

Select Program Folder画面は，スタート・メニューの確認ダイアログです．[Next>] ボタンを押すと，インストールが始まります．

(7) インストール完了

インストール完了の画面が表示されたら[Finish] ボタンで終了します．

● EDS Service Packのインストール

EDSのセットアップが終わったら，続けてService Pack 1をインストールします．

(1) ダウンロード・ファイルの実行

Altera社のWebサイトからダウンロードしたService Pack 1のセットアップ・ファイル(niosii_60_sp1_pc.exe)を起動するとWelcome画面が表示されるので，[Next>] ボタンを押します．

(2) ライセンス同意

License Agreementの画面では，内容を確認し，「I accept the terms of the license agreement」を選択して[Next>] ボタンを押します．

(3) インストール・フォルダの設定

EDSをセットアップしたフォルダが表示されます．古いバージョンのNios IIがインストールしてあると複数表示される場合があります．確認して[Next>] ボタンを押します．

(4) インストールの実行

インストール実行確認の画面が表示されたら，[Install] ボタンを押して実行を開始します．

(5) インストール完了

インストール完了の画面が表示されたら，[Finish] ボタンで終了します．

● Nios II向け開発ツール

Nios IIを使用するには，FPGA開発用のQuartus IIのほかに，いくつかの開発ツールを使用します．

(1) SOPC Builder

Altera社はSOPC Builderを，Quartus II専用のシステム統合ツールと呼んでいます．Avalonスイッチ・ファブリックというオンチップ・バスと，それに接続されるマスタおよびスレーブ・モジュールを管理するGUI(graphical user interface)ベースのツールです．

バス・マスタ(CPUコアなど)の持つアドレス空間に合わせて，スレーブ・モジュールのI/Oアドレスや割り込みの定義ができ，設定に合わせた論理ブロックを自動生成してくれます．逆に言えば，バス・マスタなしでSOPC Builderは機能しません．

スレーブ・モジュールには，シリアルI/OやパラレルI/O，タイマ，外部バス・インターフェースなど，無償で使えるIP(intellectual property)コアがあります．また，AMPP(Altera Megafunction Partners Program)企業から提供されているAvalonスイッチ・ファブリックへの直結をサポートしている有償のIPコアがあります．

Nios IIはバス・マスタとして動作するIPコアという位置づけになります．Nios IIをバス・マスタにした場合，接続したスレーブ・モジュールをアクセスするソフトウェア・ライブラリも自動生成してくれます．

自分で設計したモジュールをIPコアとして登録することも可能です．筆者は市販プロセッサとのバス・ブリッジを設計し，Nios IIを使わずにSOPC Builderでシステムを構築したことがあります．割り込みのハンドリングでひとくふう必要だったものの，けっこう便利でした．

Niosが出始めのころは，Nios専用開発環境をインストールしないと使えませんでした．しかしその後Quartus II本体に統合され，NiosそのものがSOPC Builderにアドオンされる形に変更され現在に至っています．

(2) Nios II Integrated Development Environment(IDE)

IDEはNios IIになって提供されたソフトウェア開発環境です．オープン・ソースのEclipse C/C＋＋ Development Toolsをベースに開発されています．主要機能は下記の4点です．

- プロジェクト・マネージャ
- エディタおよびコンパイラ
- デバッガ
- Flash Programmer

Flash Programmerという聞き慣れないことばが出てきましたが，これはNios IIからサポートされた機能です．FPGA外部に接続されたCFI(Common Flash Interface)準拠のフラッシュ・メモリに，JTAG経由で直接書き込みができるツールです．Flash Programmer Designという専用論理を自動生成するツールも提供されています．

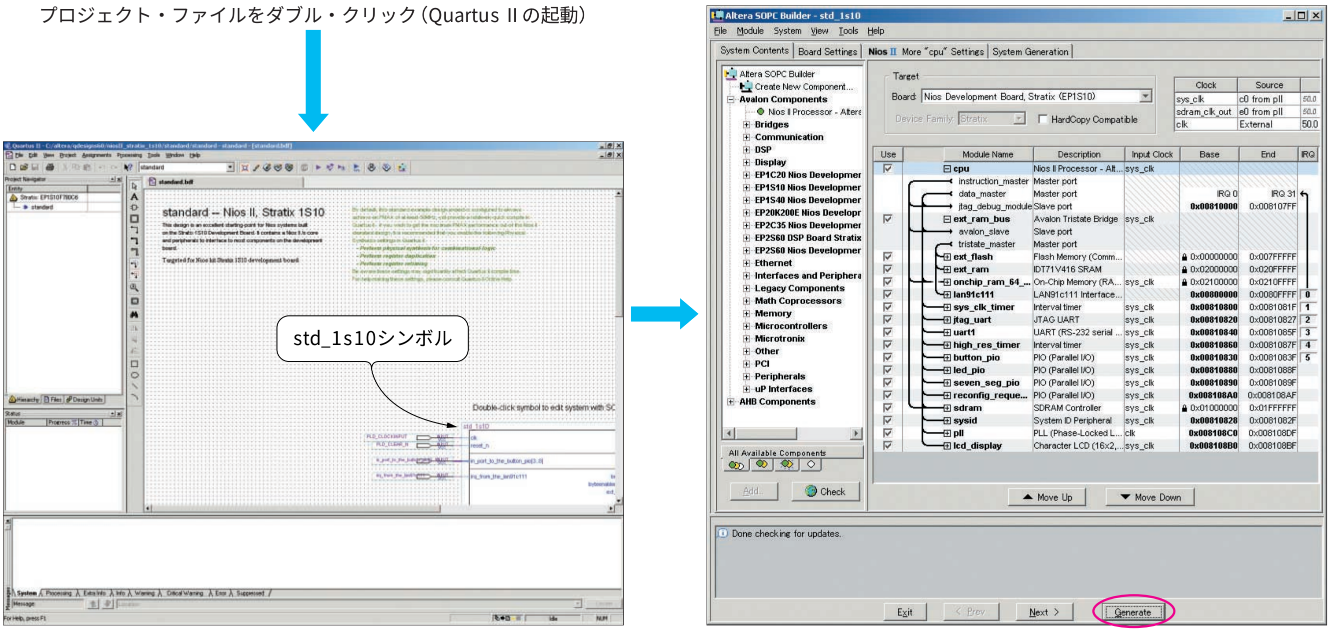


図6　Nios II回路の準備

次のページへ続く

3. Nios IIによる開発とアクセラレーション

開発環境として，今回はAltera社の「Nios II開発キットStratix エディション」を使用しました．FPGAとして，「EP1S10F780C」を搭載しています．

● Nios II回路の準備

EDSのExampleには，Nios II開発キット向けのサンプル回路が用意されています．今回は，C:\altera\kits\nios2_60\examples\vhdl（もしくはverilog）フォルダの下にあるniosII_stratix_1s10\standardをサンプル回路として使用します．C:\altera\qdesigns60\にコピーしておきます．

（1）Quartus IIを起動

サンプル回路のプロジェクトstandard.qpfをダブル・クリックします．Quartus IIが起動し，standardプロジェクトもオープンします（図6）．

（2）SOPC Builderの起動

最上位の回路図ファイルstandard.bdfを開きます．std_1s10とあるシンボルをダブル・クリックすると，SOPC Builderが起動します．

（3）SOPC Builderで回路の生成

［Generate］ボタンを押して回路を生成させます．するとSystem Generationタブへ自動的に切り替わり，論理合成にかかわるメッセージが出力されていきます．最後に生成が成功したというメッセージが出力されたのを確認し，［Exit］ボタンを押します．

（4）シンボルのアップデート

SOPC Builderを終了するとQuartus IIに戻りますが，回路の生成によりシンボルがされたので，アップデートの確認ダイアログが表示されます．ここでは［はい（Y）］ボタンを押します．次に更新するシンボルの範囲を尋ねてくるので，Selected symbol［s］ or block［s］を選択して［OK］ボタンを押します．

（5）プロジェクトのコンパイル

プロジェクトをQuartus IIでコンパイルします．「Processing」→「Start Compilation」を選択します．シンボルをアップデートした回路図ファイルを保存するかの確認で［はい（Y）］ボタンを押すと，コンパイルが始まります．正常終了のダイアログが表示されたら［OK］ボタンを押します．

（6）FPGAのコンフィグレーション

「Tools」→「Programmer」でFPGAのコンフィグレーションを行います．評価版なので，一定時間（約60分）経過すると動かなくなる旨の確認ダイアログが表示されます

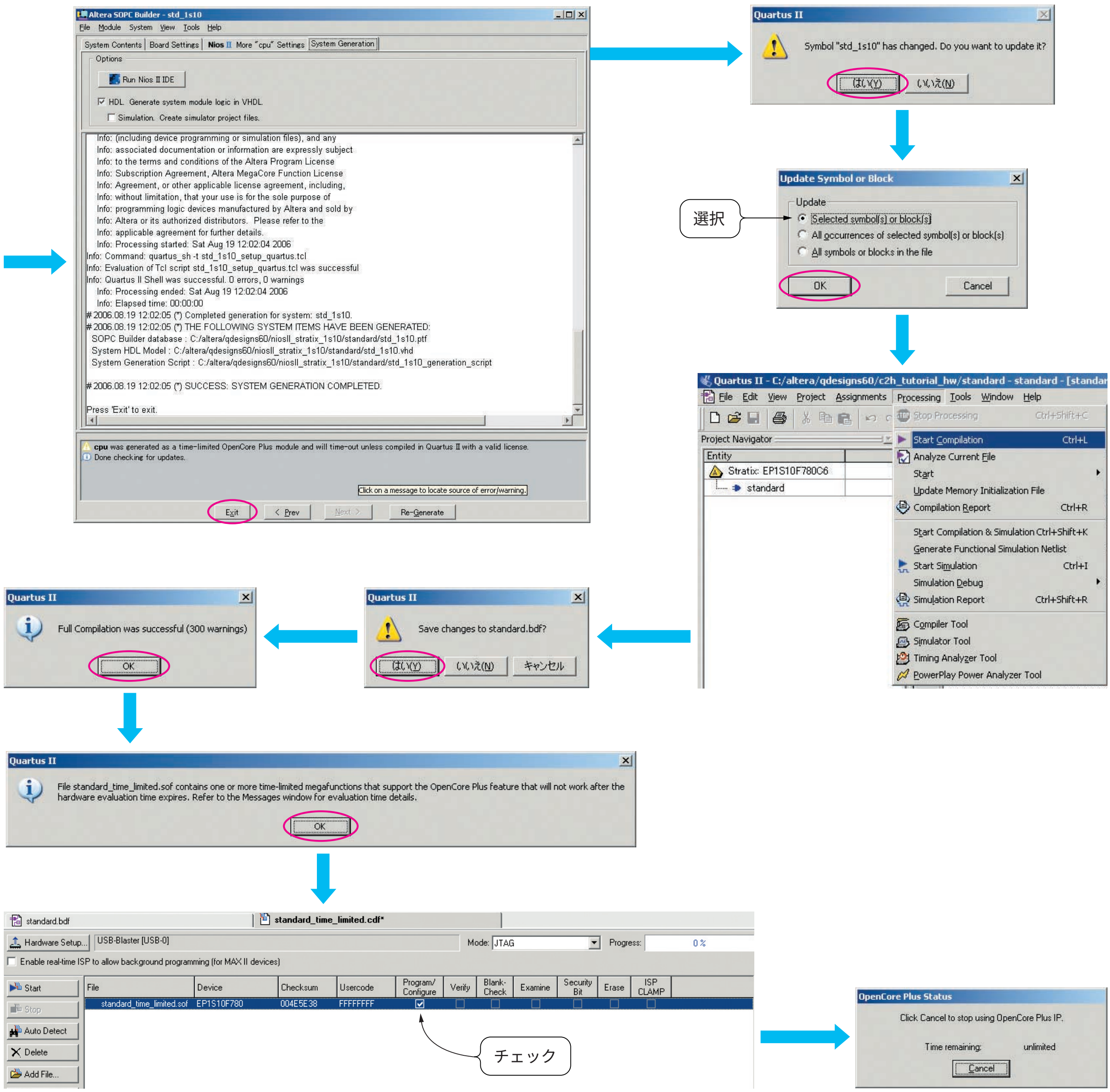


図6　Nios II回路の準備(つづき)

が，そのまま[OK]ボタンを押すとプログラマが起動します．プログラマにはデフォルトでstandard_time_limited.sofが設定されているはずです．Program/Compareにチェックを付け，[Start]ボタンを押すと，コンフィグレーションを行い，FPGAが動作を始めます．

● すべてソフトウェアとして実行

まず，サンプル・ソフトウェアをそのままNios IIで実行します．今回使用したサンプル・ソフトウェアは，Altera社のWebサイト(http://www.altera.co.jp/literature/lit-nio2.jsp)にあるdma_c2h_tutorial.cです(**リスト1**)．

mainは，do_dma関数を呼び出すだけの単純なものです．do_dma関数は，mainからソース・アドレス(*source_ptr)，デスティネーション・アドレス(*dest_ptr)，転送語数(length)の三つの引き数を受けてブロック転送を実行します．二つのアドレスはポインタなので，前述の概要から二つのバス・マスタが組み込ま

リスト1　サンプルCコード

```
#include <stdio.h>
#include <string.h>
#include <sys/alt_cache.h>
#include "sys/alt_alarm.h"

#define TRANSFER_LENGTH 1048576
#define ITERATIONS 100

int do_dma( int * __restrict__ dest_ptr, int * __restrict__ source_ptr, int length )      do_dma関数
{
  int i;      ローカル変数

  for( i = 0; i < (length / 4); i++ )
  {
    *dest_ptr++ = *source_ptr++;      ソース・アドレスからデスティネーション・アドレスへのブロック転送
  }
  return( 0 );
}

int main( void )      メイン関数
{
  int i;
  int *source_ptr, *dest_ptr;      ローカル変数
  int start_time, finish_time, total_time;

  // Buffers we'll be transferring to and from.
  source_ptr = (int*)malloc(TRANSFER_LENGTH);      メモリ領域を確保してポインタ・アドレスを得る
  dest_ptr = (int*)malloc(TRANSFER_LENGTH);

  // Fill the source buffer and erase the dest buffer
  for( i = 0; i < (TRANSFER_LENGTH / 4); i++ )
  {
    *(source_ptr + i) = i;      初期化
    *(dest_ptr + i) = 0x0;
  }

  printf("This simple program copies %d bytes of data from a source buffer to a destination buffer.¥n", TRANSFER_LENGTH);
  printf("The program performs %d iterations of the copy operation, and calculates the time spent.¥n¥n", ITERATIONS);

  printf("Copy beginning¥n");

  start_time = alt_nticks();      処理開始時のタイマ値を記録

  for( i = 0; i < ITERATIONS; i++ )
  {
    do_dma( dest_ptr, source_ptr, TRANSFER_LENGTH );      do_dma関数の呼び出し
  }

  finish_time = alt_nticks();      処理終了時のタイマ値を記録

  if( memcmp( dest_ptr, source_ptr, TRANSFER_LENGTH ) )
  {
    printf("ERROR: Source and destination data do not match. Copy failed.¥n");
  }
  else      処理の正誤判定
  {
    printf("SUCCESS: Source and destination data match. Copy verified.¥n");
  }

  free(dest_ptr);      メモリ領域を解放
  free(source_ptr);

  total_time = ((finish_time - start_time) * 1000) /
               alt_ticks_per_second();      処理時間の計算と表示

  printf("Total time: %d ms¥n", total_time);

  return 0;
}
```

れると予想できます．

（1）事前準備

ハードウェア・プロジェクトを格納しているC:¥altera¥qdesigns60¥niosII_stratix_1s10¥standardフォルダの下にsoftwareというフォルダを新規作成し，そこへdma_c2h_tutorial.cを保存します．

（2）IDEの起動

Quartus IIから，SOPC Builderを起動します．次にSystem Generationタブにある［Run Nios II IDE］ボタンを押すとIDEが起動します．

2

(3) アプリケーション・プロジェクトの新規作成

IDEのメニューから，「File」→「New」→「C/C++ Application」を実行すると，新規プロジェクト作成ダイアログが表示されます(図7)．

Name欄にプロジェクト名として「c2h_tutorial_sw」と入力します．Use Default Locationをチェックすると，Location欄にパスが設定されます．SOPC Builder System欄には，C:\altera\qdesigns60\niosII_stratix_1s10\standard\std_1s10.ptfを指定します．拡張子ptfは，SOPC Builderが出力する設定情報ファイルを意味します．Select Project Templateでは「Blank Project」を選択し，[Finish]ボタンを押します．IDEの画面にあるC/C++ Projectsタブに作成したプロジェクトが追加されていることを確認します．

(4) ソース・ファイルのインポート

IDEのメニューから，「File」→「Import」を実行すると，Importダイアログが表示されます．今回はソース・ファイルがすでにあるので，Select an import source一覧の中から，File systemを選択し[Next>]ボタンを押します．

File system画面では，From directory欄にdma_c2h_tutorial.cのパスを設定します．使用するファイルが一つしかないので，[Select All]ボタンですべてにチェックを入れます．Into folder欄にはプロジェクト名「c2h_tutorial_sw」を指定します．Optionはデフォルトのままにしておきます．

[Finish]ボタンを押すと，プロジェクトに指定したファイルがインポートされます．IDEの画面にあるC/C++ Projectsタブで，c2h_tutorial_swの前にある「+」をクリックすると，dma_c2h_tutorial.cが入っていることがわかります．さらにファイル名をダブル・クリックするとエディタが起動し，ファイルの中身が表示されます．

(5) ビルドして実機動作

IDEのメニューから「Run」→「Run As」→「1 Nios II Hardware」を実行すると，ソフトウェアのビルドと実機へのダウンロード，実行までをすべて一括で行います．IDEのConsoleタブにはその経過が逐次表示され，プログラム実行時には`printf`による出力を表示します．

実行時間(86530ms)が最後に表示され，すべてソフトウェアによるサンプル・ソフトウェアの実行確認は終了です．

● C2Hコンパイラによるハードウェア化

今回使用したサンプル・ソフトウェアでは，`do_dma`関数をハードウェア化します．この関数はポインタを使用したブロック転送なので，ソース/デスティネーションおよび転送語数を入力するスレーブ・レジスタと，二つのバス・マスタを持った構成になると予想されます．

(1) ハードウェア化対象関数の設定

dma_c2h_tutorial.cをIDEのエディタで開きます．ソース・コードの中から`do_dma`関数の定義部を見つけ，関数名(do_dma)をダブル・クリックで選択します．次に右クリックで表示されるメニューから「Accelerate with the Nios II C2H Compiler」を選択します．右下のウィンドウにC2Hタブができます（図8，pp.55-56）．

(2) ビルド・オプションの指定

C2Hタブでは，ビルドを行う際のオプションを指定します．今回は，「Build software, generate SOPC Builder system, and run Quartus II compilation」と，「Use

COLUMN　C2Hコンパイラがサポートする C 記述

Nios IIのソフトウェア・コンパイラはGNUに対し，C2Hコンパイラがサポートするのは ANSI です．GNUで書いたソース・リストはそのまま ANSI C対応のC2Hコンパイラで実行できるのかという疑問がわきます．

筆者は両コンパイラの言語解釈を細部にわたって把握しているわけではないので，「ここが違う」という明確なコメントができません．

一般的には，ANSIのほうが記述を厳密にチェックすると言われていますが，ANSIでまったく触れていない項目について，GNUのほうが詳細に解釈を記載している箇所もあると聞いています．

C2Hコンパイラは ANSI Cをフル・サポートしているわけではありません．コンパイラの言語解釈依存にならないようにソース・コードを記述しておけば，ほとんど書き換えせずにとりあえずはハードウェア化できると考えてよいのではないでしょうか．

C2Hコンパイラの制約事項にも注意が必要です．

● 変換対象はC言語のソース・ファイルのみ

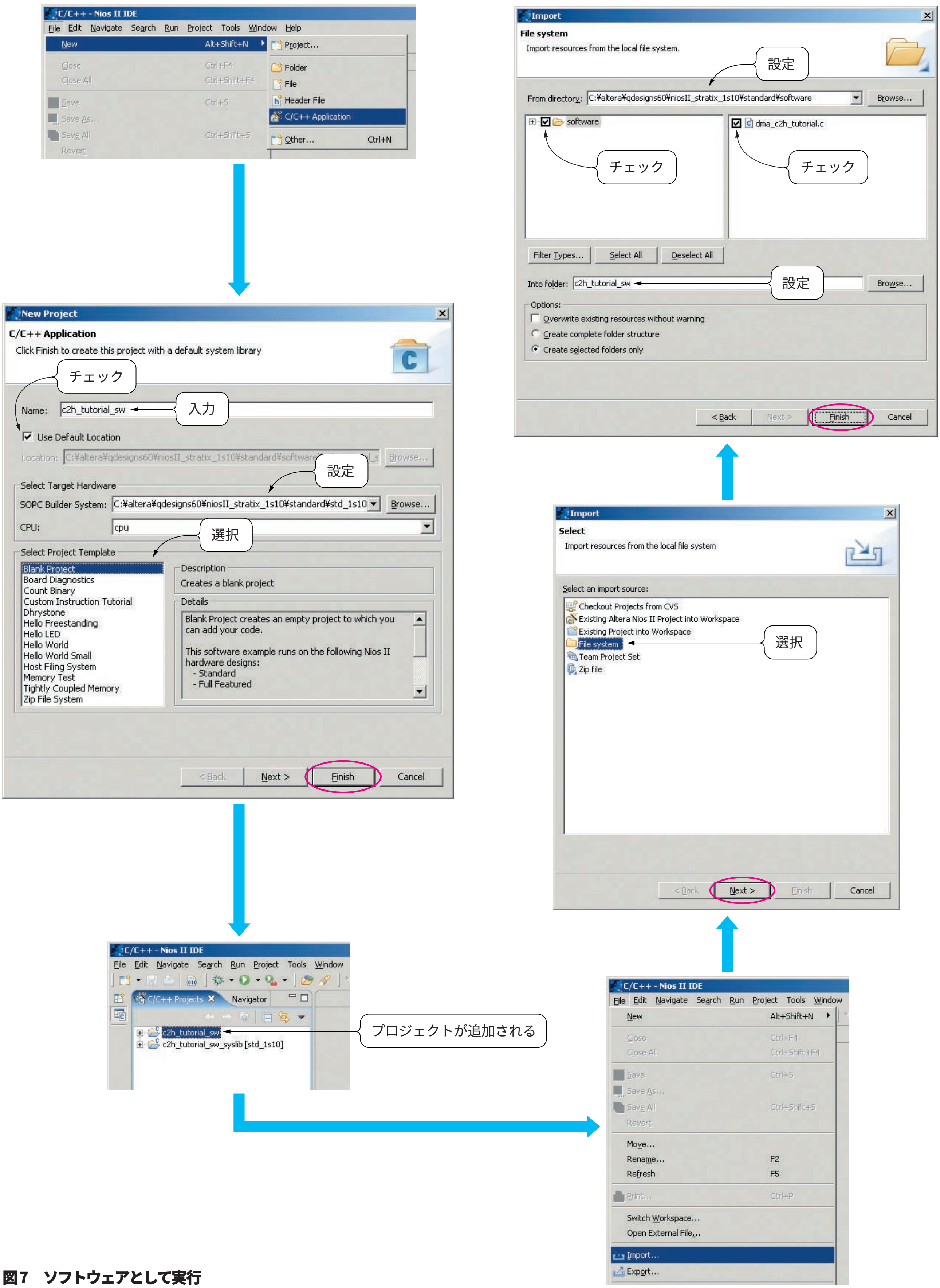
当然ですが，既存のソフトウェアでバイナリ化したライブラリを参照していると変換できません．

● mainからまるまるアクセラレートはできない

C2Hコンパイラは上位ソフトウェアからの呼び出しで起動されるハードウェアを生成するので，main関数は変換できません．もっともmainからすべてハードウェア化したら，まったく動作しないCPUを組み込むことになり，論理のむだづかいです．

● 標準入出力関数は変換できない

printfなどの標準入出力関数には対応していません．



次のページへ続く

図7　ソフトウェアとして実行

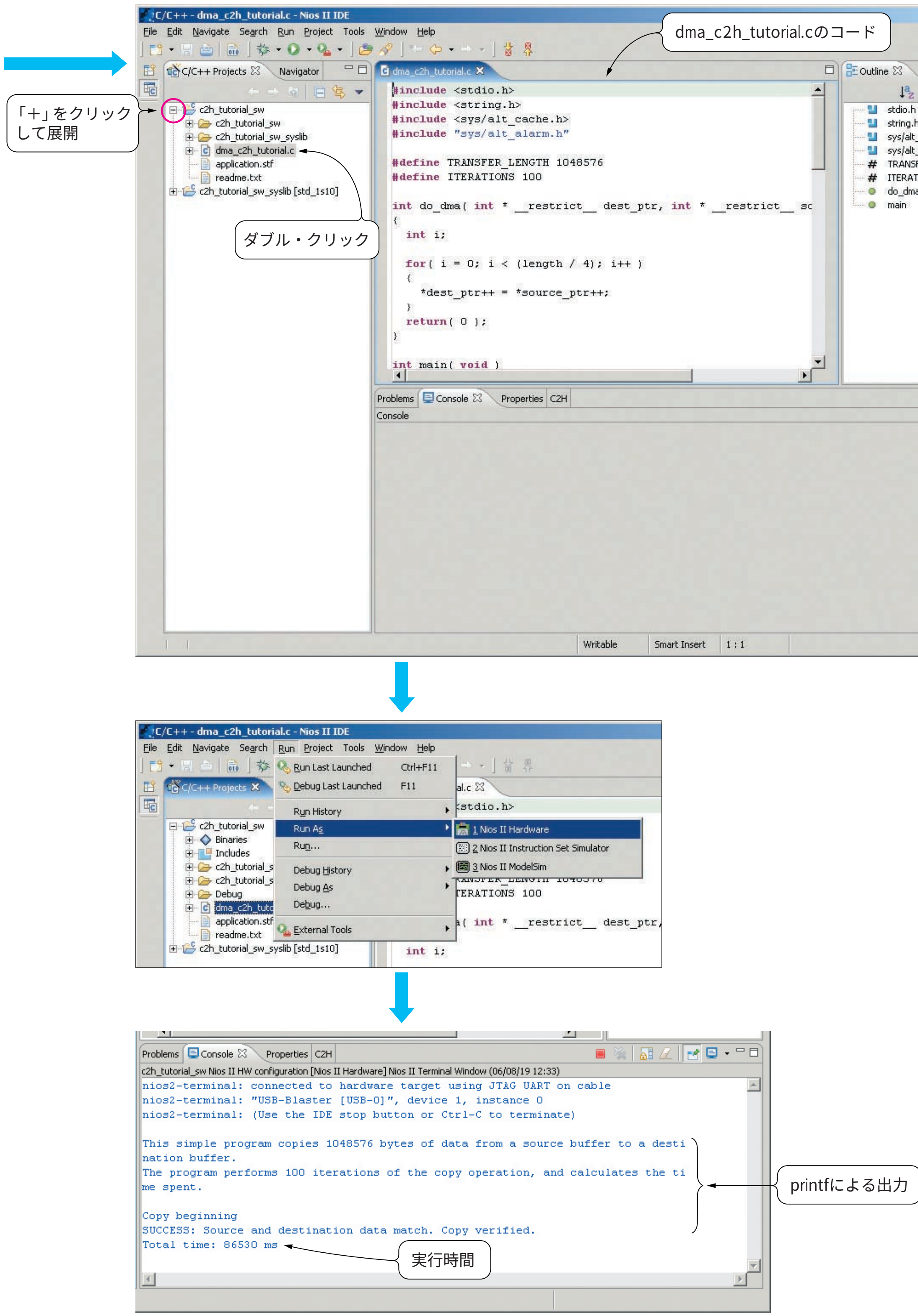


図7　ソフトウェアとして実行(つづき)

hardware accelerator in place of software implementation, Flush data cache before each call」を選択しています．つまり，IDEからのビルド・コマンドで，ソフトウェアのビルド，C2Hコンパイラによるハードウェア化，SOPC Builderへの組み込み，回路の生成，Quartus IIのコンパイルまで一気にやってしまおうという指定です．

(3) ビルド実行

IDEのメニューから「Project」→「Build Project」を実行します．C2Hコンパイラによるハードウェア化とハードウェア/ソフトウェアすべての再コンパイルを実行するので，それなりの時間がかかります．2.8GHzのPentium 4（Hyper Thread），1Gバイトのメモリを搭載するパソコン



次のページへ続く

図8　C2Hコンパイラによるハードウェア化

では，約30分かかりました．

C2Hタブには，SummaryやGlossary，Resources，Performanceのレポートが追加されます．やはり性能が気になるので，Performanceの「＋」をクリックして情報を展開表示させます．いろいろな情報がありますが，ポイントとなるのはCPLI（Cycles Per Loop Iteration）とLoop latencyの2点です．

● CPLI

for文やwhile文などのループを実行するパイプラインの最短データ入力間隔で，処理系の性能そのものを示しています．これが1に近いほど処理性能が高いことになります．サンプル・ソフトウェアの結果は1です．

● Loop latency

パイプラインのデータ入力から結果出力までのクロック数で，どんなに理想的な論理になったとしても，ループ回数が1だとCPLI＋Loop latency分だけの処理時間となります．

ハードウェア化による高速効果が大きいのはやはりループ処理です．ループ回数管理とその内部処理は並行して処理可能なことが多いためです．そのポイントをしっかりと押さえ，ツールに組み込んでいることに感心しました．

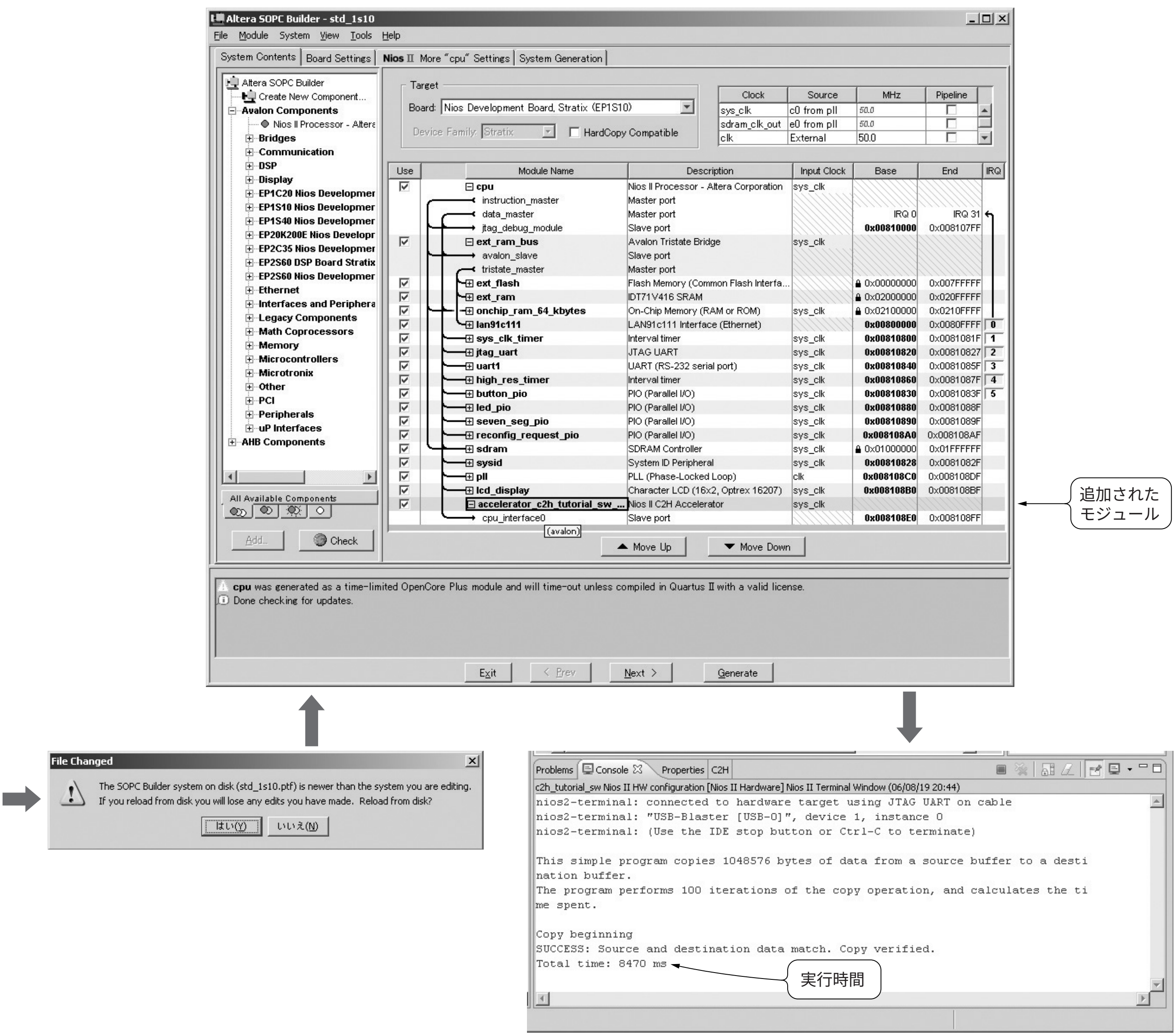


図8　C2Hコンパイラによるハードウェア化（つづき）

(4) モジュール組み込み状態の参照

SOPC Builderでハードウェア化されたモジュールの組み込み状態を参照します．

アクティブ・ウィンドウをSOPC Builderへ切り替えます．System Contentsタブを開くと，いちばん下に「accelerator_c2h_tutorial_sw ...」というモジュールが追加されていることがわかります．スレーブ・モジュールとして32バイトの領域を占有しています．

現在のバージョンでは，バス・マスタとしての接続関係は表示されません．表示が煩雑で見にくくなるためのようです．選択式でよいので，複数のCPUを使うときのように表示してほしいところです．

(5) アクセラレータの効果確認

実行速度を確認します．IDEのメニューから「Run」→「Run As」→「1 Nios II Hardware」を実行すると，ソフトウェアのダウンロード，および実行が行われます．

実行時間は8470msでした．すべてソフトウェアで実行した場合(86530ms)に比べ約10倍高速化されたことになります．

● 実際の変換内容を確認

C2Hコンパイラによる効果を確認できましたが，実際に元のソフトウェアがどのように変わったのかが気になるところです．

調べてみると，mainでdo_dma関数をコールする記述は何も変わっていませんでした．do_dma関数内が，ソフトウェアだけの実行記述から，ハードウェアをアクセスするラッパ記述に変更されていました(**リスト2**)．

C2Hコンパイラが出力したdo_dma関数のハードウェア・アクセラレータは，HDL記述でコメントも含め2,300行以上ありますので，ここでは解説しません．興味のある方は解析してみてください．

4. 評価を終えて

C2Hコンパイラは，リリースされたばかりのツールです．今後のバージョンアップでさまざまな改良がなされるであろうと考えています．そこで今回は基本機能を中心に評価しました．今後は，

2

リスト2　ハードウェアをアクセスするラッパ記述

```
/* Legal Notice: (C)2006 Altera Corporation. All rights reserved.  Your */
/* use of Altera Corporation's design tools, logic functions and other */
/* software and tools, and its AMPP partner logic functions, and any */
/* output files any of the foregoing (including device programming or */
/* simulation files), and any associated documentation or information are */
/* expressly subject to the terms and conditions of the Altera Program */
/* License Subscription Agreement or other applicable license agreement, */
/* including, without limitation, that your use is for the sole purpose */
/* of programming logic devices manufactured by Altera and sold by Altera */
/* or its authorized distributors.  Please refer to the applicable */
/* agreement for further details. */

#ifdef CHAC_THIS
#else
#include "stdio.h"
#include "io.h"
#ifndef ACCELERATOR_C2H_TUTORIAL_SW_DO_DMA_CPU_INTERFACE0_BASE
#define ACCELERATOR_C2H_TUTORIAL_SW_DO_DMA_CPU_INTERFACE0_BASE 0x008108E0
#endif

#include <sys/alt_cache.h>

__inline__ volatile  int  do_dma ( int *  __restrict__  dest_ptr,
                 int *  __restrict__  source_ptr,
                 int length
              )
{
  alt_dcache_flush_all();
  IOWR_32DIRECT(ACCELERATOR_C2H_TUTORIAL_SW_DO_DMA_CPU_INTERFACE0_BASE, (4), (int) (dest_ptr));
  IOWR_32DIRECT(ACCELERATOR_C2H_TUTORIAL_SW_DO_DMA_CPU_INTERFACE0_BASE, (8), (int) (source_ptr));
  IOWR_32DIRECT(ACCELERATOR_C2H_TUTORIAL_SW_DO_DMA_CPU_INTERFACE0_BASE, (12), (int) (length));
  /* Write 1 to address 0 starts the accelerator */
  IOWR_32DIRECT(ACCELERATOR_C2H_TUTORIAL_SW_DO_DMA_CPU_INTERFACE0_BASE,(0 * sizeof(int)),1);
  /* Poll. When read from address 0 returns 1, the accelerator is done */
  while(IORD_32DIRECT(ACCELERATOR_C2H_TUTORIAL_SW_DO_DMA_CPU_INTERFACE0_BASE,(0 * sizeof(int))) == 0)
  {}
  return (( int )IORD_32DIRECT(ACCELERATOR_C2H_TUTORIAL_SW_DO_DMA_CPU_INTERFACE0_BASE,(1*sizeof(int))));
}
#endif /* CHAC_THIS */
```

制御レジスタに'1'を書き込み処理を開始

Nios IIが持つデータ・キャッシュをフラッシュし，実メモリに書き込む

mainから渡される引き数をレジスタに書き込む

状態レジスタが'1'になるまでポーリングして処理終了を待つ

戻り値を読み出す

● C2Hコンパイラが得意/不得意とする処理
● より高い性能を得るために考慮するポイント

について，継続して評価してみたいと考えています．

● 今後のC2Hコンパイラに期待すること

組み込み機器を数多く手がけてきた筆者としては，いくつかサポートしてほしいと感じた機能がありました．元のソフトウェアを変更なしにハードウェア化できるというC2Hコンパイラの特徴にそぐわない点もありますが，一意見としてまとめてみます．

(1) 割り込みのサポート

現在のバージョンでは，アクセラレータ・モジュールからの割り込み出力をサポートしていません．ソフトウェア記述の変更なしにハードウェア化するためと推測しています．しかしせっかくハードウェアで高速化しても，処理終了までポーリングで待つのでは，たいへんもったいない気がします．

処理終了やエラー判定を割り込みで処理できればシステム性能はさらに上げられるでしょう．若干のコード修正が伴ってでもサポートしてほしい機能です．

(2) FPGA外部と直接インターフェースしたい

FPGAのI/O端子を直接制御するソフトウェアを考えます．まずハードウェアとしてI/Oポートを組み込み，それをポインタでアクセスしながら処理を行うソフトウェアを組むことになります．

このソフトウェア部をハードウェア化すると，アクセラレータ・モジュールのバス・マスタがI/Oポートのレジスタに対してアクセスを行う，いわゆるバスを介しての動作になります．ハードウェア化によりCPUからI/Oポートへのアクセスがなくなるのでバス競合は起きませんが，アクセス・サイクルはどうしても必要になります．直結できればバス制御論理およびバス・アクセスによるオーバヘッドもなくなります．

(3) カスタム命令化対応

Nios IIは，ユーザがカスタム命令を追加できます．スレーブ・インターフェースでかつ2項演算の関数は，カスタム命令に対応できればスマートです．

(4) さらなる汎用化

現在は，ハードウェア化する関数の指定は，IDEからしかできません．また，現在のSOPC Builderは，バス・マスタとしてNios IIが組み込まれていないと，ソフトウェア関連のファイルはいっさい生成されません．しかし例えば設計中のFPGA外部にCPUがあり，Avelonスイッチ・ファブリックを介して周辺モジュールを使いたい場合もあります．

Avalonスイッチ・ファブリックへ接続されるアクセラレータ・モジュールの自動生成ツールとして，より汎用的に使えるようになることを期待します．

● 雑感

C言語に限らず，一つの言語でシステム設計ができ，ハードウェアとソフトウェアの協調設計があたりまえとなる時代がいずれは来ると言われています．そうなったとき，最上位にあたるシステム設計を担当するのはハードウェア技術者，ソフトウェア技術者いずれなのでしょうか．担当職務の流れからいけば，アプリケーション側に近く人口も多いソフトウェア技術者なのでしょうが，汎用計算機上のソフトウェアとは違い，組み込み機器は「一品一様(多種多様)」であり，ハードウェア設計なしでは成り立ちません．

マルチプロセッサ構成があたりまえになり，物は小さくてもその中身は非常に大規模で複雑化しています．あちこちの製品で出荷後の不ぐあいに関する情報をよく目にする現状を考えると，市場の変化や設計技術の進歩に開発する側(人間)が追いつけなくなっているのではとも考えたりします．

システム全体について熟知し，ロードマップも含めた製品コンセプトを立案でき，開発プロジェクトもまとめきれる人材を育てることから始めるのが早道かもしれません．「急がば回れ」，「まず隗(かい)より始めよ」などと，人の道を説くことばが頭の中を巡っています．

参考・引用*文献

(1) 浅井剛，中根隆康，飛永徹；FPGAの片隅にCPUを載せてコスト削減， DesignWave Magazine，2003年8月．
(2) 藤村雅志；バージョンアップで姿を変えるソフト・マクロのCPU， DesignWave Magazine，2004年12月．
(3) Altera；Nios II Processor Reference Handbook Ver6.0， May 2006.
(4) Altera；Nios II Software Developer 's Handbook Ver6.0， May 2006.
(5) Altera；Nios II C2H Compiler User Guide Ver1.0， May 2006.

あさい・たけし
東京計器工業(株)
FPGAコンソーシアム　監事